ML SYSTEM OPTIMIZATION
Session 2

BITS Pilani

innovate achieve lead

BITS Pilani

M1: Foundations

Parallelization Paradigms

innovate achieve lead

BITS Pilani

1. *Data-level parallelism*
   Task-level parallelism
   Algorithm-specific parallelism

Parallelization

Parallelization is the decomposition of a computational workload into independent components that can execute concurrently on N processors.

Let T_{serial} be the total execution time of a task on a single processor.

If a fraction P of the task is parallelizable and is distributed across N processors, then the parallel execution time is:

$$T_{parallel} = (1 - P)T_{serial} + \frac{PT_{serial}}{N}$$

Ideal Case (No Sequential Part, No Overhead)

If the entire task is parallelizable (P=1):

$$T_{parallel} = \frac{T_{serial}}{N}$$

Speedup: $$S(N) = \frac{T_{serial}}{T_{parallel}} = N$$

*This is called **linear speedup***

Parallelization

General Case (With Overhead)

Including communication and synchronization overhead $T_{overhead}$:

$$T_{parallel} = (1 - P)T_{serial} + \frac{PT_{serial}}{N} + T_{overhead}$$

Objective of Parallelization

Minimize:

$$T_{parallel}$$

Maximize:

$$S(N) = \frac{T_{serial}}{T_{parallel}}$$

and maintain high efficiency:

$$E(N) = \frac{S(N)}{N}$$

Parallelization

Parallelization attempts to transform a computation of cost:

$$O(T)$$

into:

$$O(T / N)$$

subject to limits imposed by sequential fractions and system overhead

Work Integrated Learning Programmes

Data-Level Parallelism (DLP)

Data-Level Parallelism refers to performing the **same operation** on multiple data elements simultaneously.

It is applicable when:

- The operation on each data element is **independent**
- No inter-element dependency exists
- Data can be partitioned evenly

Given two arrays A and B of size n:

$$C[i] = A[i] + B[i],\ i = 1,2,\ldots,n$$

Each computation of C[i] is independent of others. So the total serial execution time:

$$T_{serial} = n \cdot t_{op}$$

Where:

- t_{op} = time for one addition

Data-Level Parallelism (DLP)

Parallel Execution

If N processors are available and work is evenly distributed:

$$T_{parallel} = \frac{n}{N} \cdot t_{op}$$

Speedup:

$$S(N) = \frac{T_{serial}}{T_{parallel}} = \frac{nt_{op}}{\frac{n}{N}t_{op}} = N$$

Data-Level Parallelism (DLP)

Maximum Speedup

If N = n (one processor per element):

$$T_{parallel} = t_{op}$$

$$S_{max} = n$$

In real systems:

- There is communication overhead
- Memory access delays occur
- Processors may not be perfectly synchronized

So practical speedup is slightly less than n.
Thus we write:

$$S_{max} \approx n$$

meaning **approximately linear speedup**.

Data-Level Parallelism (DLP)

Computational Complexity View

Serial Complexity

$$\textit{Serial Time Complexity} = O(n)$$

Parallel Complexity (Ideal Case, No Overhead)

$$\textit{Parallel Time Complexity} = O\,(n / N)$$

Where:

- n = problem size
- N = number of processors

If N = n

$$O\,(n / n) = 1$$

This represents **constant time complexity** in the ideal case (perfect parallelism, no communication overhead).

Data-Level Parallelism (DLP)

Image Processing

Applying a filter (e.g., blur, edge detection) to each pixel can be done independently. Each processor/thread handles a subset of pixels simultaneously.

Vector Addition / Matrix Operations

Operations such as:

$$C[i] = A[i] + B[i]$$

can be computed independently for each index i. Similarly, matrix multiplication distributes row–column computations across multiple processors.

GPU Computations

Modern GPUs such as those using **CUDA architecture by NVIDIA** allow thousands of cores to execute threads in parallel.

For example:

- Each CUDA core handles one data element.
- Large-scale computations (deep learning, simulations, rendering) achieve massive speedup.

Task-Level Parallelism (TLP)

Task-Level Parallelism (TLP) refers to executing **multiple independent tasks simultaneously** instead of executing them sequentially.

- Each task performs a different operation.
- Tasks may work on the same dataset or different datasets.
- The goal is to reduce overall completion time by overlapping execution.

Unlike data parallelism (where the same operation is applied to different data elements), TLP focuses on parallelizing **different functions or modules**.

Task-Level Parallelism (TLP)

In **TLP**, we divide the program into **independent tasks**:

$$T_1, T_2, \ldots, T_m$$

Each task performs a logically different unit of work.
Unlike data parallelism:

- Tasks may operate on **different data**
- Tasks may even execute **different algorithms**

Sequential Execution Time
If executed sequentially:

$$T_{\text{serial}} = \sum_{i=1}^{m} t_i$$

- t_i → execution time of task T_i
- Summation means tasks execute **one after another**
- Total time is additive

Task-Level Parallelism (TLP)

Assume we have m tasks:

$$T_1, T_2, \ldots, T_m$$

Each task takes time:

$$t_1, t_2, \ldots, t_m$$

Since serial execution runs tasks one by one:

$$T_{serial} = t_1 + t_2 + \cdots + t_m$$

Then,

$$T_{\text{serial}} = \sum_{i=1}^{m} t_i$$

Because:

- Task 2 starts only after Task 1 finishes.
- Task 3 starts only after Task 2 finishes.
- No overlap happens.

*So total time is the **sum of all task durations**.*

Task-Level Parallelism (TLP)

Parallel Execution Time

Now assume we have **m processors**, and each task runs on one processor.

All tasks start at the same time.

The program finishes when the **last task finishes**.

So: $T_{\text{parallel}} \approx \max(t_1, t_2, \ldots, t_m)$

Task	Time
T_1	5 sec
T_2	3 sec
T_3	2 sec

All start together.

- T_3 finishes at 2 sec
- T_2 finishes at 3 sec
- T_1 finishes at 5 sec

System completes at 5 sec. That is the **maximum time**.

*This is called the **critical path time**.*

Speedup in Task-Level Parallelism (TLP)

Speedup measures how much faster the parallel system is compared to serial execution.

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

- T_{serial} → Total time taken when tasks are executed one after another.
- $T_{parallel}$ → Total time taken when tasks are executed simultaneously.

Why Serial divided by Parallel?: Because we want to measure improvement.

$$\text{Speedup} = \frac{\text{Old Time}}{\text{New Time}}$$

- Old time → Serial time
- New time → Parallel time

If:

T_{serial} = 100 seconds

$T_{parallel}$ = 20 seconds

Then:

S = 100 / 20 = 5

*The parallel system is **5 times faster**.*

Work Integrated Learning Programmes

Load Balance Ratio in Task-Level Parallelism (TLP)

Average Task Time

$$\bar{t} = \frac{1}{m} \sum_{i=1}^{m} t_i$$

- m → number of tasks
- t_i → execution time of task i
- $\bar{t}$ → average execution time per task

If total work is divided equally among tasks, each task should ideally take $\bar{t}$ time.

It represents the **ideal balanced workload per processor**.

Load Balance Ratio in Task-Level Parallelism (TLP)

Ideal Parallel Time

$$T_{\text{ideal}} = \bar{t}$$

If:

- All tasks are perfectly balanced
- Each processor gets equal work

Then every task finishes at the same time.

So total completion time = average task time.

*This is the **best possible parallel time** (assuming no overhead).*

Actual Parallel Time

$$T_{parallel} = max(t_i)$$

Even if most tasks are small:

- The slowest task determines when the system finishes.
- Faster processors must wait for the slowest one.

*This is the **bottleneck effect**.*

Task-Level Parallelism (TLP)

When Number of Tasks ≠ Number of Processors

m = number of tasks
p = number of processors
If m > p → All tasks cannot execute simultaneously, so tasks must be ***scheduled***

Parallel Time with Scheduling

$$T_{\text{parallel}} = \max_{j=1}^{p} \left(\sum t_i \text{ assigned to processor } j \right)$$

- Each processor gets multiple tasks
- Total time per processor = sum of assigned task times
- Overall completion time = maximum workload among processors

When m>p, performance depends more on intelligent scheduling than on the number of processors.

Task-Level Parallelism (TLP)

Adding Overhead

$$T_{parallel} = max(t_i) + T_{overhead}$$

- $max(t_i)$ → Longest task (critical path)
- $T_{overhead}$ → Extra system costs

Overhead Includes

- Communication delay
- Synchronization (locks / barriers)
- Task or thread creation
- Context switching

Speedup with Overhead

$$S = \frac{\sum t_i}{\max(t_i) + T_{\text{overhead}}}$$

Algorithm-Specific Parallelism (ASP)

Algorithm-Specific Parallelism refers to designing **parallel algorithms tailored to the mathematical structure of a specific problem**.

- Unlike generic data-level or task-level parallelism, ASP **exploits the algorithm's inherent recursion, decomposition, or structure** for concurrency.
- Often used in **scientific computing, numerical methods, and signal processing**.

Instead of simply dividing tasks or data, we analyze:

- Recurrence relations
- Decomposition structure
- Algebraic independence
- Divide-and-conquer properties

*Parallelism is extracted from the **theory of the algorithm itself**.*

Parallel Sorting

Serial Merge Sort

Recurrence relation:

$$T_{serial}(n) = 2\,T(n/2)+O(n)$$

- Divide array in two halves
- Sort each half sequentially
- Merge sorted halves in O(n) time

Parallel Merge Sort

If two halves are sorted **simultaneously on 2 proc**

$$T_{\text{parallel}}(n) = \frac{2\,T(n/2)}{N} + O(n)$$

Where:

- N = number of processors used for the recursive step
- Merge step still requires O(n) time

Parallel Sorting

Suppose N=2, array size n=8:

1. Divide → 2 subarrays of size 4, sorted **concurrently**
2. Each subarray further divides → recursion continues
3. Merge step at each level takes O(n)

Parallelization reduces the **depth of recursion** execution time

Speedup is limited by the **merge step** (sequential part)

This is an example of **Amdahl's law applied recursively**.

Work Integrated Learning Programmes

Matrix Multiplication

Divide A and B into submatrices (blocks)

$C = A \cdot B$

Compute block products independently:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Each block multiplication is independent → execute in parallel

Strassen Algorithm (Parallel): Strassen reduces number of multiplications using recursion

$$T_{\text{serial}}(n) = 7\,T(n/2) + O(n^2)$$

Parallel form:

$$T_{\text{parallel}}(n) = \frac{7\,T(n/2)}{N} + O(n^2)$$

- N processors handle recursive multiplications simultaneously
- Significant speedup, especially on GPUs

Algorithm-Specific Parallelism (ASP)

Most algorithm-specific parallel algorithms follow:

$$T_{\text{parallel}}(n) = \frac{\text{Recursive Subproblems}}{N} + T_{\text{combine}}$$

Where:

- Recursive subproblems → can be executed concurrently
- $T_{combine}$ → sequential portion for merging or summing results

Speedup:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \leq \frac{T_{\text{serial}}}{T_{\text{combine}} + T_{\text{parallel-subtasks}}}$$

- Limited by **sequential combine step** (Amdahl's Law)

Work Integrated Learning Programmes

Algorithm-Specific Parallelism (ASP)

Algorithm-specific parallelism **maximizes concurrency** by exploiting the problem's structure

Algorithm	Parallel Strategy
Merge Sort	Sort halves concurrently
FFT	Recursive even-odd decomposition
Matrix Multiplication	Block multiplication / Strassen
Numerical PDE Solvers	Domain decomposition

Efficiency depends on:

- Number of processors N
- Sequential combine time
- Task granularity

Work Integrated Learning Programmes

Parallelization Paradigms

Paradigm	Concept	Example	Hardware Best Suited
Data-Level Parallelism (DLP)	Same operation on multiple data elements	Vector addition, image filter	GPUs, SIMD units
Task-Level Parallelism (TLP)	Different tasks executed concurrently	Web server requests, pipeline	Multi-core CPUs, clusters
Algorithm-Specific	Parallel algorithms for problem structure	Parallel merge sort, FFT	CPU cores, GPUs, clusters